Dynamic Timeouts and Neighbor Selection Queries in Peer-to-Peer Networks

Wolfgang Hoschek
CERN IT Division
European Organization for Nuclear Research
1211 Geneva 23, Switzerland
wolfgang.hoschek@cern.ch

ABSTRACT
In a Peer-to-Peer (P2P) network, non-pipelining query result set delivery without a dynamic abort timeout feature is highly unreliable due to what we propose as the *simultaneous abort problem*: If only one of the many nodes in the query path fails to be responsive for whatever reasons, all other nodes in the chain are waiting, eventually time out at the same time, and the originator receives not even a single result. To address the problem, we derive *dynamic abort timeouts* using as policy *exponential decay with halving*. This ensures that a maximum of results can be delivered reliably within the time frame desired by a user. We establish that a timeout for loop detection in query routes must be static. A dynamic timeout is unsuitable to be used as *loop timeout*, due to what we propose as the *non-simultaneous loop timeout problem*.

In a P2P network, a node forwards a query to the set of nodes obtained from *neighbor selection*. Using neighbor selection, explicit topology characteristics can be exploited in query guidance. The best neighbor selection policy to adopt depends on the context of the query and the topology. For flexibility and expressiveness, we propose to allow the user to specify the selection policy. In addition to the normal query, the user defines a *neighbor selection query* (XQuery) that takes the tuple set of the current node as input and returns a subset that indicates the nodes selected for forwarding. A wide range of policies can be implemented in this manner, as the neighbor selection policy can draw from the rich set of information contained in the tuples published to the node.

KEY WORDS
Peer-to-Peer Networks, Messaging, Service Discovery

1. Introduction

In a large distributed system such as a Peer-to-Peer (P2P) file sharing system [1, 2] or a Grid [3], it is desirable to maintain and query dynamic and timely information about active participants such as services, resources and user communities. Other examples are a (worldwide) service discovery infrastructure for a multi-national organization, the Domain Name System (DNS), the email infrastructure, a monitoring infrastructure for a large-scale cluster of clusters, or an instant messaging and news service. For example, the European DataGrid (EDG) [4, 5, 6] is a software infrastructure that ties together a massive set of globally distributed organizations and computing resources for data-intensive physics analysis applications, including thousands of network services, tens of thousands of CPUs, WAN Gigabit networking as well as Petabytes of disk and tape storage [7].

An enabling step towards increased Internet and Grid software execution flexibility is the *web services* vision [4, 8, 9] of distributed computing where programs are no longer configured with static information. Rather, the promise is that programs are made more flexible and powerful by querying Internet databases (registries) at runtime in order to discover information and network attached third-party building blocks. Services can advertise themselves and related metadata via such databases, enabling the assembly of distributed higher-level components.

In support of this vision we have introduced the *Web Service Discovery Architecture (WSDA)* [10] and given motivation and justification [11] for the assertion that realistic ubiquitous service and resource discovery requires a rich general-purpose query language such as XQuery [12] or SQL [13]. Based on WSDA, we introduced the *hyper registry* [14], which is a centralized database (node) for discovery of dynamic distributed content.

However, in an Internet discovery database system, the set of information tuples in the universe is partitioned over one or more distributed nodes (peers), for reasons including autonomy, scalability, availability, performance and security. It is not obvious how to enable powerful discovery query support and collective collaborative functionality that operate on the distributed system as a whole, rather than on a given part of it. Further, it is not obvious how to allow for search results that are fresh, allowing time-sensitive dynamic content.

It appears that a Peer-to-Peer (P2P) database network may be well suited to support dynamic distributed database search, for example for service discovery. The overall P2P idea is as follows. Rather than have a centralized database, a distributed framework is used where there exist one or more autonomous database nodes, each maintaining its own data. Queries are no longer posed to a central database; instead, they are recursively propagated over the network to some or all database nodes, and results are collected and send back to the client.

Consequently, we devised the WSDA based *Unified Peer-to-Peer Database Framework (UPDF)* [4] and its associated *Peer Database Protocol (PDP)* [15], which are unified in the sense that they allow to express specific applications for a wide range of data types (typed or untyped XML, any MIME type [16]), node topologies (e.g. ring, tree, graph), query languages (e.g. XQuery, SQL), query response modes (e.g. Routed, Direct and Referral Response) and pipelining characteristics. In the UPDF framework, an *originator* sends a query to an *agent* node, which evaluates it, and forwards it to select *neighbor nodes*.

A *link topology* describes the link structure among nodes. For example, in a worldwide service discovery system, a link topology can tie together a distributed set of administrative domains, each hosting a registry node holding descriptions of services local to the domain. Several link topology models covering the spectrum from centralized models to fine-grained fully distributed models can be envisaged, among them single node, star, ring, tree, semi hierarchical as well as graph models. Figure 1 depicts some example topologies.

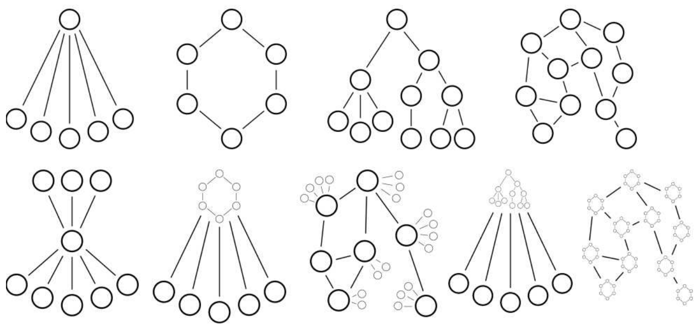

Figure 1. Example Link Topologies [17].

In this paper, we answer the following questions:

- *How can a maximum of results be delivered reliably within the time frame desired by a user, even if a query type does not support pipelining? How can loops be detected reliably using timeouts? How can flexible neighbor selection policies be used to exploit topology characteristics in answering a query?*

Non-pipelining delivery without a dynamic abort timeout feature is highly unreliable due to the so-called *simultaneous abort problem*: If only one of the many nodes in the query path fails to be responsive for whatever reasons, all other nodes in the chain are waiting, eventually time out at the same time, and the originator receives not even a single result. To address the problem, we propose *dynamic abort timeouts* using as policy *exponential decay with halving*. This ensures that a maximum of results can be delivered reliably within the time frame desired by a user. A dynamic timeout is unsuitable to be used as *loop timeout*, due to the non-simultaneous loop timeout problem. A loop timeout must be static.

A node forwards a query to the set of nodes obtained from *neighbor selection*. The best neighbor selection policy to adopt depends on the context of the query and the topology. For example, a query may only select neighbors that meet minimum requirements in terms of latency and bandwidth. Using neighbor selection explicit topology characteristics can be exploited in query guidance. For flexibility and expressiveness, we propose to allow the user to specify the selection policy. In addition to the normal query, the user defines a *neighbor selection query* (XQuery) that takes the tuple set of the current node as input and returns a subset that indicates the nodes selected for forwarding. A wide range of policies can be implemented in this manner. The neighbor selection policy can draw from the rich set of information contained in the tuples published to the node.

This paper is organized as follows. Section 2. describes timeouts for loop detection in query routes and for query abort. Section 3. discusses flexible neighbor selection policies. Section 4. compares our work with existing research results. Finally, Section 5. concludes this paper.

2. Loop and Abort Timeout

Clearly there comes a time when a user is no longer interested in query results, no matter whether any more results might be available. The query roaming the network and its response traffic should fade away after some time. In addition, P2P systems are well advised to attempt to limit resource consumption by defending against *runaway* queries roaming forever or producing gigantic result sets, either unintended or malicious. To address these problems, an absolute *abort timeout* is attached to a query, as it travels across hops. An abort timeout can be seen as a deadline. Together with the query, a node tells a neighbor *"I will ignore (the rest of) your result set if I have not received it before 12:00:00 today."* The problem, then, is to ensure that a maximum of results can be delivered reliably within the time frame desired by a user.

The value of a *static timeout* remains unchanged across hops, except for defensive modification in flight triggered by runaway query detection (e.g. infinite timeout). In contrast, it is intended that the value of a *dynamic timeout* be decreased at each hop. Nodes further away from the originator may time out earlier than nodes closer to the originator.

2.1 Dynamic Abort Timeout

A static abort timeout is entirely unsuitable for non-pipelined result set delivery, because it leads to a serious reliability problem, which we propose to call *simultaneous abort timeout.* If just one of the many nodes in the query path fails to be responsive for whatever reasons, all other nodes in the path are waiting, eventually time out and attempt to return at least a partial result set. However, it is impossible that any of these partial results ever reach the originator, because all nodes time out *simultaneously* (and it takes some time for results to flow back). For example, the agent times out and attempts to return its local partial results to the originator. After that, all partial results flowing to the agent from neighbors, and their neighbors, etc. are discarded – it is already too late. However, even the agent cannot deliver results to the originator because the originator has already timed out (shortly) before the results arrive. Hence, the originator receives not even a single result if just one of the many nodes in the query path fails to be responsive.

To address the simultaneous abort timeout problem, we propose dynamic abort timeouts. Under *dynamic abort timeout*, nodes do not time out at the same time. Instead, nodes further away from the originator time out earlier than nodes closer to the originator. This provides some safety time window for the partial results of any node to flow back across multiple hops to the originator. Together with the query, a node tells a neighbor *"I will ignore (the rest of) your result set if I have not received it before 12:00:00 today. Do whatever you think is appropriate to meet this deadline"*. Intermediate nodes can and should adaptively decrease the timeout value as necessary, in order to leave a large enough time window for receiving and returning partial results subsequent to timeout.

Observe that the closer a node is to the originator, the more important it is (because if it cannot meet its deadline, results from a large branch are discarded). Further, the closer a node is to the originator, the larger is its response and bandwidth consumption. Thus, as a good policy to choose the safety time window, we propose *exponential decay with halving*. The window size is halved at each hop, leaving large safety windows for important nodes and tiny window sizes for nodes that contribute only marginal result sets. Also, taking into account network latency and the time it takes for a query to be locally processed, the timeout is updated at each hop `N` according to the following recurrence formula:

$$timeout_N = currtime_N + \frac{timeout_{N-1} - currtime_N}{2} \tag{1}$$

Consider for example Figure 2. At time `t` the originator submits a query with a dynamic abort timeout of `t+4` seconds. In other words, it warns the agent to ignore results after time `t+4`. The agent in turn intends to safely meet the deadline and so figures that it needs to retain a safety window of 2 seconds, already starting to return its (partial) results at time `t+2`. The agent warns its own neighbors to ignore results after time `t+2`. The neighbors also intend to safely meet the deadline. From the 2 seconds available, they choose to allocate 1 second,

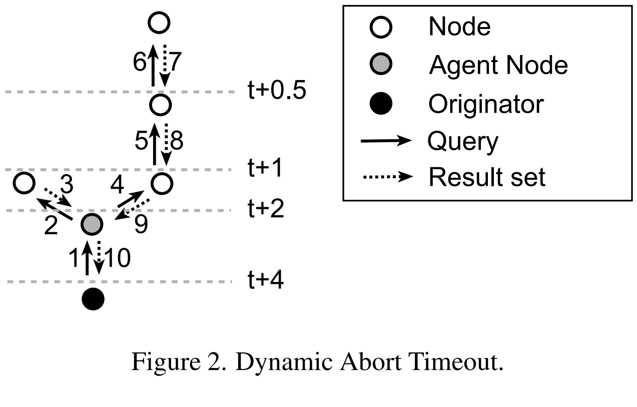

Figure 2. Dynamic Abort Timeout.

and leave the rest to the branch remaining above. Eventually, the safety window becomes so small that a node can no longer meet a deadline on timeout. The results from the unlucky node are ignored, and its partial results are discarded. However, other nodes below and in other branches are unaffected. Their results survive and have enough time to hop all the way back to the originator before time `t+4`.

Instead of ignoring results which miss their deadline a node may also close the connection. This may, but need not, be harmless. The connection is typically simply reestablished as soon as a new query is to be forwarded. However, in an attempt to educate good P2P citizens, a node may choose to stop propagating or deny service to neighbors that repeatedly do not meet abort deadlines. For example, a strategy may use an exponential back-off algorithm. Note that as long as a node obeys its timeout it can independently implement any timeout policy it sees fit for its purposes without regard to the policy implemented at other nodes. If a node misbehaves or maliciously increases the abort timeout, it risks not being able to meet its own deadline, and is likely soon dropped or denied service. Such healthy measures move less useful nodes to the edge of the network where they cause less harm, because their number of topology links tends to decrease.

To summarize, under non-pipelined result set delivery, dynamic abort timeouts using *exponential decay with halving* ensure that a maximum of results can be delivered reliably within the time frame desired by a user. We speculate that dynamic timeouts could also incorporate sophisticated cost functions involving latency and bandwidth estimation and/or economic models.

2.2 Static Loop Timeout

Interestingly, a static loop timeout is required in order to fully preserve query semantics. A dynamic timeout (e.g. the dynamic abort timeout) is unsuitable to be used as loop timeout. Otherwise, a problem arises that we propose to call *non-simultaneous loop timeout.* The same query may arrive at a node multiple times, along distinct routes, perhaps in a complex pattern. Loops in query routes must be detected and prevented. Otherwise, unnecessary or endless multiplication of workloads would be caused. To this end, a node maintains a state table of recent transaction identifiers and associated *loop timeouts* and returns an error whenever a query is received that has already been seen (according to the state table). Before the loop timeout is reached, the same query can potentially arrive multiple times, along distinct routes. On loop timeout, a node may "forget" about a query by deleting it from the state table. To be able to reliably detect a loop, a node must not forget a transaction identifier before its loop timeout has been reached.

However, let us assume for the moment that a dynamic timeout (e.g. the dynamic abort timeout) is used as loop timeout. Consider for example, Figure 3, which is identical to Figure 2 except that the agent has an additional neighbor that can potentially receive the query along more than one path. At time `t` the originator submits a query with a dynamic abort timeout of `t+4` seconds. The agent in turn warns its own neighbors to ignore results after time `t+2`. Request 5 is sent and arrives, is processed, and its results (step 8) are delivered before the dynamic abort timeout of time `t+1`. At time `t+1` the loop timeout is reached and the query is deleted from the state table. For many reasons, including temporary network segment problems and sequential neighbor processing, request 10 can be delayed. In the example it arrives after time `t+1`. By this time, the receiving node has already forgotten that it already handled the very same query. Hence, the node cannot detect the loop and continues to process and forward (step 11, 12) the same query again.

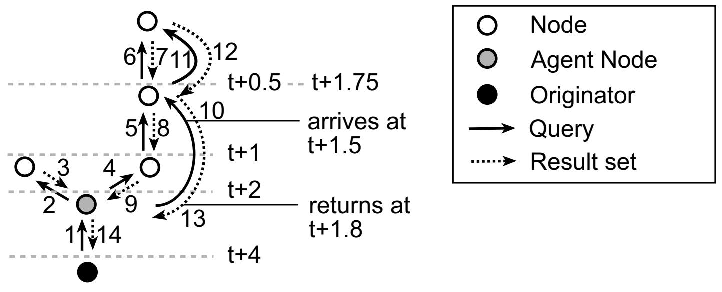

Figure 3. Loop Detection Failure with Dynamic Loop Timeout.

The non-simultaneous loop timeout problem is caused by the fact that some nodes still forward the query to other nodes when the destinations have already forgotten it. In other words, the problem is that loop timeout does not occur simultaneously everywhere. Consequently, a loop timeout must be static (does not change across hops) to guarantee that loops can reliably be detected. Along with a query, an originator not only provides a dynamic abort timeout, but also a static loop timeout. Initially at the originator, both values must be identical (e.g. `t+4`). After the first hop, both values become unrelated.

To summarize, we have `abort timeout` $\leq$ `loop timeout`. Loop timeouts must be static whereas abort timeouts may be static or dynamic. Under non-pipelined result set delivery, dynamic abort timeouts using *exponential decay with halving* ensure that a maximum of results can be delivered reliably within the time frame desired by a user. A dynamic abort timeout model still requires static loop timeouts to ensure reliable loop detection, so that a node does not forward and answer the same query multiple times.

3. Neighbor Selection Query

For simplicity, all our discussions so far have implicitly assumed a *broadcast* model (on top of TCP) in which a node forwards a query to all neighbor nodes. However, in general one can select a subset of neighbors, and forward concurrently or sequentially. Fewer query forwards lead to less overall resource consumption. The issue is critical because of the snowballing (epidemic, flooding) effect implied by broadcasting. Overall bandwidth consumption grows exponentially with the query radius, producing enormous stress on the network and drastically limiting its scalability. For details, consult [18, 19].

Clearly selecting a neighbor subset can lead to incomplete coverage, missing important results. The best policy to adopt depends on the context of the query and the topology. Context is *required* to improve on the broadcast model. For example, it makes little sense to forward a Gnutella query to non-Gnutella nodes. The scope can select only neighbors with a service description of interface type "Gnutella". In an attempt to explicitly exploit topology characteristics, a virtual organization of a Grid may deliberately organize global, intermediate and local job schedulers into a tree-like topology. Correct operation of scheduling may require reliable discovery of all or at least most relevant schedulers in the tree. In such a scenario, random selection of half of the neighbors at each node is certainly undesirable. A policy that selects all `child` nodes and ignores all `parent` nodes may be more adequate.

Further, a node may maintain statistics about its neighbors. One may only select neighbors that meet minimum requirements in terms of latency, bandwidth or historic query outcomes (`maxLatency`, `minBandwidth`, `minHistoricResult`). Other node properties such as hostname, domain name, owner, etc. can be exploited in query scope guidance, for example to implement security policies. Consider an example where the scheduling system may only trust nodes from a select number of security domains. Here a query should never be forwarded to nodes not matching the trust pattern.

Further, in some systems, finding a single result is sufficient. In general, a user or any given node can guard against unnecessarily large result sets, message sizes and resource consumption by specifying the maximum number of result tuples (`maxResults`) and bytes (`maxResultsBytes`) to be returned. Using sequential propagation, depending on the number of results already obtained from the local database and a subset of the selected neighbors, the query may no longer need to be forwarded to the rest of the selected neighbors.

For maximum result set size limiting, a timeout and/or radius can be used in conjunction with neighbor selection, routed response, and perhaps sequential forward, to implement the *expanding ring* [20] strategy. The term stems from IP multicasting. Here an agent first forwards the query to a small radius/timeout. Unless enough results are found, the agent forwards the query again with increasingly large radius/timeout values to reach further into the network, at the expense of increasingly large overall resource consumption. On each expansion radius/timeout are multiplied by some factor.

For flexibility and expressiveness, we propose to allow the user to specify the selection policy. In addition to the normal query, the user defines a *neighbor selection query* (XQuery) that takes the tuple set of the current node as input and returns a subset that indicates the nodes selected for forwarding. For example, a neighbor query implementing broadcasting selects all services with registry and P2P query capabilities, as follows:

```
RETURN /tupleset/tuple[@type="service"
  AND content/service/interface[@type="Consumer"]
  AND content/service/interface[@type="XQuery"]]
```

A wide range of policies can be implemented in this manner. The neighbor selection policy can draw from the rich set of information contained in the tuples published to the node. Further, recall that the set of tuples in a database may not only contain service descriptions of neighbor nodes (e.g. in WSDL [21]), but also other kind of (soft state) content published from any kind of content provider. For example, this may include the type of queries neighbor nodes can answer, descriptions of the kind of tuples they hold (e.g. their types), or a compact summary or index of their content. Content available to the neighbor selection query may also include host and network information as well as statistics that a node periodically publishes to its immediate neighbors. A neighbor selection query enables group communication to all nodes with certain characteristics (e.g. the same group ID or interfaces). One can implement domain filters and security filters (e.g. `allow/deny` regular expressions as used in the Apache HTTP server if the tuple set includes metadata such as hostname and node owner. To summarize, a neighbor selection query can be used to implement *smart dynamic routing*.

As usual, for security reasons, a node may choose to ignore, override or extend any scope hints it receives. The neighbor query concept can also be used for flexible policy implementation internal to a node. In this case, a node always ignores the user provided neighbor query and uses an internal custom neighbor selection query instead.

4. Related Work

Radius. The *radius* of a query is a measure of path length. More precisely, it is the maximum number of hops a query is allowed to travel on any given path. The radius is decreased by one at each hop. The roaming query and response traffic must fade away upon reaching a radius of less than zero. The radius helps to limit latency and bandwidth consumption and to guard against runaway queries with infinite lifetime. In Gnutella [1] and Freenet [2], the radius is the primary means to specify a query scope. The radius is termed *TTL (time-to-live)* in these systems. Neither of these systems support timeouts and flexible neighbor selection.

Loop Detection. The X.500 protocol [22] uses a route-tracing algorithm for loop detection. This algorithm only works for queries that select on a name from a hierarchical name space that is mimicked by the link topology. In our general context, the algorithm is insufficient for loop detection because we allow, but do not assume, such a topology and namespace. The route-tracing algorithm attaches to a query the route already taken, represented by a list of node identifiers. On query forward, a node N appends its own identifier to the route. A loop is detected if the identifier of the current node N is already contained in the route. This mechanism only detects a loop if a query forwarded by a given node N eventually arrives again at the same node N. It cannot detect the more common form of loop where the same query arrives along multiple paths at a given node N, but none of the paths have so far touched N.

Neighbor Selection. *Iterative deepening* [23] is a similar technique to *expanding ring* where an optimization is suggested that avoids reevaluating the query at nodes that have already done so in previous iterations. Neighbor selection policies that are based on randomness and/or historical information about the result set size of prior queries are simulated and analyzed in [24]. An efficient neighbor selection policy is applicable to simple queries posed to networks in which the number of links of nodes exhibits a power law distribution (e.g. Freenet and Gnutella) [25]. Here most (but not all) matching results can be reached with few hops by selecting just a very small subset of neighbors (the neighbors that themselves have the most neighbors to the n-th radius). Note, however, that the policy is based on the assumption that not all results must be found and that all query results are equally relevant. Depending on the application context, this assumption may or may not be valid. These related works discuss in isolation neighbor selection techniques for a particular query type, without the context of a framework for comprehensive query support.

DNS. Distributed databases with a hierarchical name space such as the Domain Name System (DNS) [26] can efficiently answer queries of the form *"Find an object by its full name"*. These systems arrange the link topology, according to the hierarchical name space, as a tree topology. A query searching for the IP address of a domain name traverses the tree on the shortest path from originator to the node containing the domain name - first up, then down. At each node, a *name resolution* policy selects the neighbor "closer" to the name than the current node, according to name space metadata. In DNS, queries are not forwarded (routed) through the topology. Instead, a node returns a *referral* message that redirects an originator to the next closer node. The originator explicitly queries the next node, is referred to yet another closer node, and so on. The DNS referral behavior can be implemented within our framework by using a radius scope of zero. The same holds for the LDAP referral behavior (see below).

X.500, LDAP and MDS. The hierarchical distributed X.500 directory [22] works similarly to the DNS. It also supports referrals, but in addition can forward queries through the topology (*chaining* in X.500 terminology). The query language is simple [4]. Route tracing is used as a loop detection algorithm. Query scope specification can support maximum result set size limiting. It does not support radius and dynamic abort timeout as well as pipelined query execution across nodes. LDAP [27] is a simplified subset of X.500. Like DNS, it supports referrals but not query forwarding. The Metacomputing Directory Service (MDS) [28, 29] inherits all properties of LDAP. MDS additionally implements a simple form of query forwarding that allows for multi-level hierarchies but not for arbitrary topologies. Here neighbor selection forwards the query to LDAP servers overlapping with the query name space. The query is forwarded "as is", without loop detection. Further, MDS does not support radius and dynamic abort timeout, pipelined query execution across nodes as well as direct response and metadata responses.

5. Conclusions

There comes a time when a user is no longer interested in query results, no matter whether any more results might be available. The query roaming the network and its response traffic should fade away after some time. The value of a *static timeout* remains unchanged across hops. In contrast, it is intended that the value of a *dynamic timeout* be decreased at each hop. Nodes further away from the originator may time out earlier than nodes closer to the originator. Non-pipelined delivery with a static abort timeout is highly unreliable due to the so-called simultaneous abort timeout problem. To address the problem, we propose *dynamic abort timeouts* using as policy *exponential decay with halving*. This ensures that a maximum of results can be delivered reliably within the time frame desired by a user. A dynamic timeout is unsuitable to be used as *loop timeout*, due to the non-simultaneous loop timeout problem. A loop timeout must be static.

A query scope is used to navigate and prune the link topology and filter on attributes of the deployment model. Conceptually, the scope is the input fed to the query. One can indirectly specify a scope based on neighbor selection, timeout, radius and result set properties. A node forwards a query to the set of nodes obtained from *neighbor selection*. The best neighbor selection policy to adopt depends on the context of the query and the topology. For example, a query may only select neighbors that meet minimum requirements in terms of latency and bandwidth. Using neighbor selection explicit topology characteristics can be exploited in query guidance. For flexibility and expressiveness, we propose to allow the user to specify the selection policy. In addition to the normal query, the user defines a *neighbor selection query* (XQuery) that takes the tuple set of the current node as input and returns a subset that indicates the nodes selected for forwarding. A wide range of policies can be implemented in this manner. The neighbor selection policy can draw from the rich set of information contained in the tuples published to the node.

References

[1] Gnutella Community. Gnutella Protocol Specification v0.4. dss.clip2.com/GnutellaProtocol04.pdf.

[2] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Workshop on Design Issues in Anonymity and Unobservability*, 2000.

[3] Ian Foster, Carl Kesselman, and Steve Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *Int'l. Journal of Supercomputer Applications*, 15(3), 2001.

[4] Wolfgang Hoschek. *A Unified Peer-to-Peer Database Framework for XQueries over Dynamic Distributed Content and its Application for Scalable Service Discovery*. PhD Thesis, Technical University of Vienna, March 2002.

[5] Ben Segal. Grid Computing: The European Data Grid Project. In *IEEE Nuclear Science Symposium and Medical Imaging Conference*, Lyon, France, October 2000.

[6] Wolfgang Hoschek, Javier Jaen-Martinez, Asad Samar, Heinz Stockinger, and Kurt Stockinger. Data Management in an International Data Grid Project. In *1st IEEE/ACM Int'l. Workshop on Grid Computing (Grid'2000)*, Bangalore, India, December 2000.

[7] Large Hadron Collider Committee. Report of the LHC Computing Review. Technical report, CERN/LHCC/2001-004, April 2001. http://cern.ch/lhc-computing-review-public/Public/Report_final.PDF.

[8] Ian Foster, Carl Kesselman, Jeffrey Nick, and Steve Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration, January 2002. http://www.globus.org.

[9] P. Cauldwell, R. Chawla, Vivek Chopra, Gary Damschen, Chris Dix, Tony Hong, Francis Norton, Uche Ogbuji, Glenn Olander, Mark A. Richman, Kristy Saunders, and Zoran Zaev. *Professional XML Web Services*. Wrox Press, 2001.

[10] Wolfgang Hoschek. The Web Service Discovery Architecture. In *Proc. of the Int'l. IEEE/ACM Supercomputing Conference (SC 2002)*, Baltimore, USA, November 2002. IEEE Computer Society Press.

[11] Wolfgang Hoschek. A Data Model and Query Language for Service Discovery. Technical report, DataGrid-02-TED-0409, April 2002.

[12] World Wide Web Consortium. XQuery 1.0: An XML Query Language. *W3C Working Draft*, December 2001.

[13] International Organization for Standardization (ISO). Information Technology-Database Language SQL. *Standard No. ISO/IEC 9075:1999*, 1999.

[14] Wolfgang Hoschek. A Database for Dynamic Distributed Content and its Application for Service and Resource Discovery. In *Int'l. IEEE Symposium on Parallel and Distributed Computing (ISPDC 2002)*, Iasi, Romania, July 2002.

[15] Wolfgang Hoschek. A Unified Peer-to-Peer Database Protocol. Technical report, DataGrid-02-TED-0407, April 2002.

[16] N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. *IETF RFC 2045*, November 1996.

[17] Nelson Minar. Peer-to-Peer is Not Always Decentralized. In *The O'Reilly Peer-to-Peer and Web Services Conference*, Washington, D.C., November 2001.

[18] Jordan Ritter. Why Gnutella Can't Scale. No, Really. http://www.tch.org/gnutella.html.

[19] Matei Ripeanu. Peer-to-Peer Architecture Case Study: Gnutella Network. In *Int'l. Conf. on Peer-to-Peer Computing (P2P2001)*, Linkoping, Sweden, August 2001.

[20] S.E. Deering. *Multicast Routing in a Datagram Internetwork*. PhD Thesis, Stanford University, 1991.

[21] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. *W3C Note 15*, 2001. http://www.w3.org/TR/wsdl.

[22] International Telecommunications Union. Recommendation X.500, Information technology – Open System Interconnection – The directory: Overview of concepts, models, and services. *ITU-T*, November 1995.

[23] Beverly Yang and Hector Garcia-Molina. Efficient Search in Peer-to-Peer Networks. In *22nd Int'l. Conf. on Distributed Computing Systems*, Vienna, Austria, July 2002.

[24] Adriana Iamnitchi and Ian Foster. On Fully Decentralized Resource Discovery in Grid Environments. In *Int'l. IEEE Workshop on Grid Computing*, Denver, Colorado, November 2001.

[25] L. Adamic, R. Lukose, A. Puniyani, and B. Huberman. Search in power-law networks. *Phys. Rev*, E(64), 2001.

[26] P. Mockapetris. Domain Names - Implementation and Specification. *IETF RFC 1035*, November 1987.

[27] W. Yeong, T. Howes, and S. Kille. Lightweight Directory Access Protocol. *IETF RFC 1777*, March 1995.

[28] Karl Czajkowski, Steven Fitzgerald, Ian Foster, and Carl Kesselman. Grid Information Services for Distributed Resource Sharing. In *Tenth IEEE Int'l. Symposium on High-Performance Distributed Computing (HPDC-10)*, San Francisco, California, August 2001.

[29] Steven Fitzgerald, Ian Foster, Carl Kesselman, Gregor von Laszewski, Warren Smith, and Steven Tuecke. A Directory Service for Configuring High-Performance Distributed Computations. In *6th Int'l. Symposium on High Performance Distributed Computing (HPDC '97)*, 1997.